Automation of Data Traffic Control on DSM Architecture

Michael Frumkin, Haoqiang Jin, Jerry Yan [1]
Numerical Aerospace Simulation Systems Division
NASA Ames Research Center

Abstract

Design of distributed shared memory (DSM) computers liberates user from the duty to distribute data across processors and allows incrementally develop parallel programs using, for example, OpenMP or Java threads. DSM architecure greatly simplifies development of parallel programs having good performance on a few processors. However, to achieve a good program scalability on DSM computers requires from the user to understand data flow in the application and use various techniques to avoid data traffic congestions. In this paper we discuss a number of such techniques, including data blocking, data placement, data transposition and page size control and evaluate their efficiency on the NAS Parallel Benchmarks. We also present a tool which automates detection the constructs causing data congestions in Fortran array oriented codes and advises the user on code transformations for improving data traffic in the application.

1 Significance of Data Traffic Control

There are very few explicit programming constructs which allow to express data location in the computer memory [2]. As a result, the data location depends on the compiler and OS and the time for accessing data varies with the computer architecture and the user has to reconsider performance of his code on a machine with new architecture. The simple traps in memory access such as cache trashing and false sharing are not difficult to identify and to avoid with array dimensions padding and variables privatization. The other data traffic problems such as poor data locality, excessive TLB misses and thread interference are difficult to diagnose and cure. Many programming constructs looking similar may have factor 3-4 difference in performance. Even the best implementations of CFD codes achieve only about 20% of peak performance of the machine it runs on (see [13]), spending 80% of time accessing data.

Several factors contribute to low efficiency of CFD codes. First, the balance of the number of floating point operations does not allow to provide an optimal mixture of instructions to keep all functional units busy. Second, and a larger factor comes from the fact that the many operations are stalled during computation waiting for data, see Example 1 in Section 2 and Table 1.

Approach to data traffic optimization. The first step in addressing this challenge is to detect the code constructs suffering from data congestions and identify the data congestion

[1] M/S T27A-2, NASA Ames Research Center, Moffett Field, CA 94035-1000; e-mail: {frumkin,hjin,yan}@nas.nasa.gov

[2] Under explicit constructs we mean such statements as "register" qualifyer in C or data distribution directives in HPF.

type causing loss of the performance. We use four congestion metrics in this paper: Primary Data Cache (PDC) misses, Secondary Cache (SC) misses, Translation Lookaside Buffer (TLB) misses, and Cache Invalidations (CI). The second step is to choose a proper remedy to resolve congestions and apply them to the code. This remedy can be either code transformation, or changing program environment such as page size or page placement/migration mechanisms.

Methods for controlling data traffic. In a number of publications these problems have been addressed [4, 9, 10] and a few techniques have been developed to improve data traffic. These techniques include data grouping for cache optimization, optimization of the page size for reduction of TLB misses, placement of pages to resolve contention in memory channel and data transposition for reduction of cache invalidations and TLB misses. Even with these techniques in hand it is not easy for nonexpert in the target computer architecture to identify where and how the appropriate transformations should be applyed. For example, it is known for some time that computations of `rhsz` and `zsolve` in BT and SP of NAS Parallel Benchmarks (OpenMP version) have factor of 3-4 worse performance then corresponding operators in x-direction, cf. [7]. Neither reason for this nor way for fixing the problem have been known so far in spite of that NPB de facto are standard codes for reporting improvements in compilers and tools [11].

Many optimizations to improve data traffic are implemented in the compilers for the target architecture. These optimizations include nest level optimizations such as loop interchange, loop fusion, software pipelining, prefetching and others. On the data level compilers usually use padding and privatization. Compilers, however, cannot perform deep code analysis such as full interprocedural or dependency analysis. Many types of analysis are not possible at compile time at all.

Another approach to identify data traffic problems relies on using hardware counters which allow to collect statistics of events during execution of a program construct. Some tools for collecting and analyzing the events, including `perfex` and `Performeter` have been built on the top of these counters [6, 12]. These tools allow to instrument the code and identify the code constructs with anomalies. However these tools are diagnostic in nature and leave analysis of the problems and searching ways to resolve them to the user.

In this paper we present a tool which identifies data traffic problems and suggests solutions for resolving them. This tool analyses data-to-data affinity and data-to-computations affinity in the code and informs the user about possible problems with the data traffic. The data control statements which can not be evaluated at the compile time the tool inserts in the code. These statements are evaluated and the user receives warnings about poor performing code constructs at run time.

We demonstrate the tool's ability to identify problematic data traffic constructs and to suggest a cure of the performance problems on three simulated CFD applications BT, SP and LU of NAS Parallel Benchmarks. The tool was able to resolve data traffic problems with `rhsz` and `zsolve` operators and helped to improve performance of the codes by 27% in average.

2 Automaton of Detection of Data Traffic Problems

For controlling data traffic the user has to have information on data movement across computer memory hierarchy in his application. Details of such movement can be complicated and machine-dependent and may require expertise in the target computer architecture. In many cases however data movement can be formulated in terms of cache parameters, array offsets, and data access strides and characterized by simple metrics such as cache misses and cache invalidations. In a few cases, such as accessing shared data, the data traffic depends on cache coherency protocol and is sensitive to the variations in the order of the execution of statements by different threads.

The requirements to the user awareness of the specifics of the computer architecture could be greately reduced with a help of a tool detecting data traffic congestions and advising ways to resolve them. Such a tool can advise on data grouping for avoiding data streaming through the processor by increasing data reuse, on initial data placement for avoiding contention in the memory access, on choosing an optimal page size for reducing the number of TLB misses and on reducing thread interference caused by cache coherence issues.

The typical problem with the code which tool intended to advise is shown in the following example.

Example 1. The first two nests in `zsolve` of SP from NAS Parallel Benchmarks (serial version) are shown in Figure 1, left pane. The optimization of the computations in the first nested loop actually slows down computations since it touches a large number of memory pages and has a poor utilization of primary cache, see `lhsz` curve in Figure 2. Merging the first and the second nested loops and recalculation of the expressions, see Figure 1, right pane, decreases the number of pages accessed, improves cache utilization, and the total execution time in spite of increase in the total number of floating point instructions, see Figure 2 `lhsz_t` curve.

We have implemented such a tool by adding features to ADAPT (Automatic Data Alignment and Placement Tool), see [2]. Originally ADAPT was designed for automatic annotating FORTRAN code with HPF directives [3]. The tool is able to identify data-to-data and to data-to-computations affinity and express the affinities through HPF ALIGN and DISTRUBUTE directives. The ability of the tool to extract data-to-data affinity and data-to-computations affinity is the key for enabling it with automatic data traffic control capabilities.

Data-to-data affinity. Two data are affine if both are used at the same instruction executed during the program run. For a pair of arrays used in the same loop nest statement the affinity relation is a correspondence between array elements referred with the same value of the loop index. Grouping affine data items together and organizing groups into a continuous stream often improve the program performance by hiding the memory latency. In general, the affinity relation is a many-to-many relation and there are many ways to group affine data items. In [3] it is shown that the possibility of grouping affine array elements depends on the geometry of the self interference lattice of the array that is a set of solutions of the Cache Miss Equation [4].

[3] ADAPT is built on the top of CAPTools [8]. It uses a CAPTools generated data base, CAPTools code analysis and some CAPTools utilities.

lhsz

```
do j=1,ny
  do i=1,nx
   do k=1,nz
    cv(k)=ws(i,j,k)
    rhon(k)=SFunction(rho(i,j,k))
   end do
   do k=1,nz
    lhs(i,j,k,1)=0.0d0
    lhs(i,j,k,2)=-dttz2*cv(k-1)
                 -dttz1*rhon(k-1)
    lhs(i,j,k,3)= 1.0d0
                 +c2dttz1*rhon(k)
    lhs(i,j,k,4)= dttz2*cv(k+1)
                 -dttz1*rhon(k+1)
    lhs(i,j,k,5)=0.0d0
   end do
  end do
end do
```

lhsz_t

```
   do k=1,nz
do j=1,ny
  do i=1,nx
    lhs(i,j,k,1)=0.0d0
    lhs(i,j,k,2)=-dttz2*ws(i,j,k-1)
       -dttz1*SFunction(rho(i,j,k-1))
    lhs(i,j,k,3)=1.0d0
       +c2dttz1*SFunction(rho(i,j,k))
    lhs(i,j,k,4)=dttz2*ws(i,j,k+1)
       -dttz1*SFunction(rho(i,j,k+1)
    lhs(i,j,k,5)=0.0d0
  end do
end do
   end do
```

Figure 1: *Data Traffic Optimization.* Original code (left) taken from `lhsz.f` of NPB2.3-serial, saves few floating point instructions used in `SFunction`. Such loop ordering creates a large number of TLB misses since calculations scan through many memory pages and a large number of PDC misses since it uses only one word per cache line. By rearranging computations (right pane) this problems are resolved improving execution time in spite of increase in number of FPI. The profiles of both codes are shown in Figure 2.

The affinity relation can be deduced for each pair of arrays in each nest statement. A control dependence results in affinity relations between the arrays involved in the control statement and all arrays in the basic blocks immediately dominated by the statement. The most common case we observe in CFD applications is one-to-many affinity relations between arrays resulted from difference operators on structured discretization grids. These relations can be approximated by a stencil (i.e. by a set of vectors with constant elements) and we call them *stencil relations.* In order to deduce the affinity for arrays used in different statements of the same nest ADAPT uses the chain rule, see [2]. The chain rule allows to propagate an affinity relation along each directed path in the nest data flow graph. The union of these relations over all directed paths leading to an array q from an array u forms the nest affinity relation between q and u. The relation lists all elements of u used for computation of each element of q and is one-to-many mapping.

Data-to-computations affinity. We represent program by a bipartite graph called program affinity graph. Let C be the set of program statements, and let D be the program data, i.e. the set of memory locations referenced in the program. We say a memory location d is affine to a statement c if the datum at address d is either operand or result of c. The program affinity graph has C and D as the vertices of the parts and an arc connecting each statement with data affine to it. Many program properties can be expressed in terms of the affinity

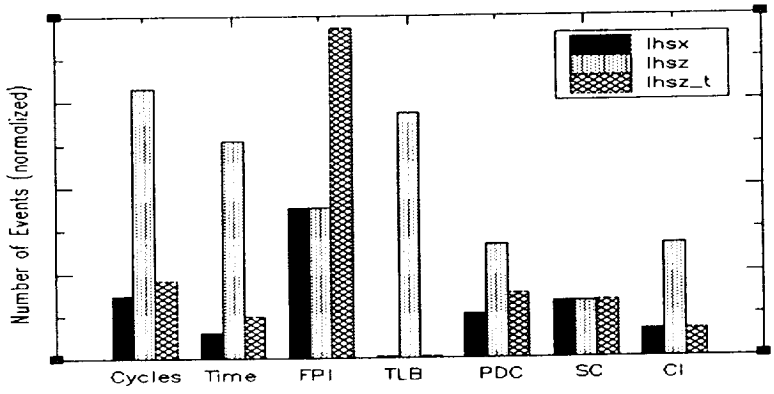

Figure 2: The effect of TLB (Table Lookaside Buffer) and PDC (Primary Data Cache) optimization on the performance of `lhsz` nest. The performance of `lhsx` nest is given as a reference. The horizontal axis shows different types of events measured with the use of hardware counters. The vertical axis shows a normalized number of measured events. Here FPI stands for Floating Point Instructions, SC stands for Secondary Cache misses, and CI stands for the secondary Cache Invalidations.

graph. For example, a statement $c2$ depends on a statement $c1$ if there is a direct path from $c1$ to $c2$. Otherwise, $c1$ and $c2$ are independent and can be executed in any order.

The analysis of the affinity graph can be simplified by indexing the statements inside the nests and the memory locations used by the arrays. In this case the arcs connecting data and statements can be expressed as a pair of expressions $(I; \texttt{idx}(I))$ where I is a vector loop index and $\texttt{idx}(I)$ is a memory address of an array element referenced at iteration I. In most of the cases in our application domain (CFD applications on structured grids) the index function is linear function of I with symbolic coefficients known at compile time. The are few nests in our applications where this is not a case. These nests include the core of the FFT algorithm where the $\texttt{idx}(i,j,k) = i + 2 \cdot j \cdot k$ for kji loop nest; nests working with multiple grids where `idx` function is read from a file; nests working with specially enumerated grid points use `idx` function stored in a precomputed array. The tool indicates the nests with nonlinear access functions without any further analysis of the nests at this point. In most nests with the linear access function the coefficients of the matrix representing the idx function are elements of the set {-1,0,1}, with an exclusion of the multigrid methods where the coefficients are multiple of 2.

Some properties of the data traffic can be deduced using only symbolic information on the coefficients of `idx` function (see the thread noninterference condition below) others require knowledge of the actual numerical values of the coefficients (see the subsection on generation of interference free tiles). If the property of the data traffic can be expressed in a symbolic form but can not be verified without knowing the numerical values of the coefficients the tool inserts the expression in the code and the user obtains the warning at run time. We call such test *run time test.*

Checking cache unfriendly access patterns. In general, cache friendly computations involve good temporal and spatial locality [5] and can not be expressed in simple terms [4]. However, some necessary conditions for cache friendly computations can be formulated and

checked. The first condition is simple: the coefficient at the innermost loop index is 1. Otherwise, nonunit stride in memory access can cause underutilization of data loaded into the cache. The other two conditions, self and cross array interference are formulated below.

Detection of self interference. If the self affinity relation is a stencil relation the tool represents the addresses of the corresponding array elements as a polylinear functions of array sizes and the index coefficients. Then for each pair of the stencil vectors it generates a set of constraints for the array dimensions in the form $nx \cdot ny \neq k \cdot S$, where nx, ny are the array sizes, S is a cache size and k is a small integer. If neither nx nor ny is known at compile time a test for a satisfiability of these constraints is inserted into a program as a run time test.

Detection of cross interference. The cross interference between two arrays happens when affine elements are mapped to the same cache location. The detection of the cross interference is similar to the detection of self interference with a difference that it involves the inter array offset and dimensions of both arrays. The cross interference constraints are represented by a polylinear inequality: $nxa \cdot nya + nxb \cdot nyb + off\_a\_b \neq k \cdot S,\ k = 1, 2, 3$. An evaluation of this inequality requires knowledge of *off_a_b* and can be done, for example, if both arrays are in the same common block or are redimensioned areas of the same bigger array.

Detection of high TLB misses. Large number of TLB misses (as in Example 1) usually results from large memory stride due to iterations of the innermost loop of a nest. Our TLB miss test checks two conditions: 1. the number of iterations of the innermost nest exceeds the TLB_SIZE; 2. the distance between the first an last address accessed in the innermost loop exceeds the PAGE_SIZE*TLB_SIZE. If both conditions can be proved to be true then the user gets a warning about high TLB misses in the nest. Otherwise, if both conditions can not be proved to be false then the tool inserts a run time test.

Checking thread noninterference. This condition can be formulated as nonoverlapping of the address spaces accessed by different threads [4]. If the noninterference condition is satisfied then the memory accessed by a thread can be placed at the memory of the processor running the thread, improving data locality. This condition is checked only for "read/write" arrays since thread interference would cause cache lines invalidations. In the case of "read" array this condition is not checked since read arrays are copied into secondary cache anyway and one thread do not affect others.

Consider a single nest of a parallel program and assume that the parallelized loop (k-loop in this case) is known. Let an array access function be a linear function of the nest indices i, j, k with symbolic coefficients a, b, c:

$$addr_p(i, j, k) = ai + bj + ck + cwp$$

where p is the thread number, w is the number of the loop iterations per thread, $0 \leq i < nx, 0 \leq j < ny, 0 \leq k < w, 0 \leq p < P$, P is the number of threads. Then the necessary and sufficient condition for thread noninterference simply is

$$c > a(nx - 1) + b(ny - 1).$$

[4] That is if a thread accesses array elements at addresses A and B then no other thread accesses an array element at address C, if $A \leq C \leq B$.

If there are multiple array access functions per array then the tool checks the noninterference condition for each function. This, however, is not sufficient for complete thread noninterference, for example. If the access functions differ by a constant term (independent on i, j, k) the threads have small interference.

Interference of threads depends on the data sharing protocol implemented in the DSM computer. For example, if data coherence is supported on the level of secondary cache lines, as in the Origin 2000, then read/write interference can happen even if two threads are accessing two different words of the same cache line. For a tool it is possible to be aware of the data sharing protocol and adjust CI estimates accordingly. We have implemented a more general approach: for each nest and each thread the tool evaluates an interference indicator as a ratio of the number of memory locations adjacent to the memory locations accessed by other threads to the total number of memory locations accessed by a the thread. For example, in Figure 1 the interference ratio for `lhs` is $P/nx \cdot ny$ in `lshz` and $P/nx \cdot ny \cdot nz$ in `lshz_t`, which correlates well with the CI number in Figure 2. This interference indicator is similar to "surface-to-volume" ratio used to estimate cache utilization, and to communications-to-computations ratio in MPI programs.

Detection of the data sources and the initial data placement. The page placement on a DSM computer commonly is controlled by one of simple policies such as "First Touch" or "Round Robin". More sophisticated page placements can be implemented with special tools such as `dplace` (see [12]). Incorrect initial data placement, for example, concentration of data at a single processor, can cause memory contention at the execution time and hamper the application scalability. By this reason we enabled the tool by an ability to detect data initialization constructs in the code. We found that all data initialization constructs in our codes have one of following types:

- reading data from a file
- initialization of arrays from another array
- initialization of an array with an intrinsic function

These constructs are easily detectable by our tool and a data placement directive (in the form of HPF ALIGN, DISTRIBUTE directives) is issued before each construct.

3 Experiments

As experimental platform we used an SGI Origin 2000 installed at NASA Ames Research Center. We conducted experiments on 16 processors of 512 node 400 MHz machine. We submitted jobs through the Portable Batch System (PBS) which dedicates requested resources to the job and minimizes it interference with other jobs running on the machine. The execution time variation between runs was within 1% indicating that PBS provided a good isolation from other jobs and a consistent mapping of the application onto the machine. We used 16 processors since this is the minimal number of processors where the slowdown due to memory traffic effects was well pronounced for the the 64^3 grids of NPB class A. The effects are similar up to 32 processors, after which the parallelization limits become dominant.

The primary memory hierarchy on Origin 2000 involves 4 levels: registers, primary data and instruction caches, secondary unified data and instruction cache, and the main memory. The access time to data located on different levels of memory is shown in Table 1, cf. [12].

The primary cache is 2-way associative having 512 lines of 32 bytes long in each set. The

Table 1: Access Time (in machine cycles) to Data in Origin 2000K Memory.

Data Location	Access Time	Condition
registers	0	
L1 cache	2-3	L1 hit
L1 cache	8-10	L1 miss, L2 hit
L2 cache	75-250	L2 miss, TLB hit
L2 cache	~2000	L2 miss, TLB miss

secondary cache is shared by data and instructions and it is also 2-way associative having 32K 128 bytes long lines. The main memory is split in chunks of 512 MB between nodes (2 processors per node) totaling 128 GB of memory on 512 processor machine. TLB has 64 entires containing base addresses of 64 pairs of pages.

The cache coherency protocol guarantees that data accessed by different processors do not stale. The protocol invalidates a line in secondary cache every time any processor requests an exclusive ownership (usually for writing) of data mapped to the line. In this case all copies in all other processors are invalidated and each processor working with this line has to request a fresh copy of the line to resume computations. An obvious implication of this protocol is that it will cause significant slowdown if two processors would attempt to write data located within the same 128 byte segment of main memory.

As the test codes we chose OpenMP version of PBN3.0-b2 a release of NAS Parallel Benchmarks which includes optimized serial, OpenMP, HPF and Java versions of the benchmarks. PBN (which stands for Programming Baseline for NPB) is designed for demonstrating capabilities of the compilers and tools working with CFD codes [1].

Me measured execution time of various levels of optimized programs. Since we used `-O3` flag to the compiler then in many cases a special effort required to prevent the compiler from undoing our optimizations. We provided flags for compiler for suppressing prefetching: `--LNO:prefetch=0` to obtain an accurate numbers for the harware counters, and flag `OPTFLAGS=-OPT:reorg_common=OFF` to enforce our own padding of arrays declared on the common blocks.

4 Experimental Results and Discussion

We applied the tool to SP, BT and LU of NAS Parallel Benchmarks [4] (optimized OpenMP version PBN-O). For each benchmark the tool was able to generate the following data traffic optimization diagnostics.

• *Nests for initial data placements.* The tool detected all nests where data were initialized. In all cases the arrays were initialized from array of smaller dimensions (or constants). The initial data placement was appropriately implemented in the original code and we did not do any relevant changes.

• *Nests with nonunit strides and advise on loop interchange.* Such nests were detected only in calculation of the right hand side array by the subroutines `rhs`, `exact_rhs`, and `erhs`.

• *Nests with big strides and advise on loop interchange and data transpositions.* Nests with big strides were detected in the subroutines `rhs`, `exact_rhs`, `erhs` and in `zsolve`. An advise on interchanging *jik* to *kji* loops were issued for the first three subroutines. In `zsolve` a dependency in k index did not allow to parallelize the loop on k and make a loop interchange.

• *Nests with self or cross interference, and advises on padding.* No arrays with self interference were detected (the existing paddings of the second and third dimensions were sufficient). The cross interference condition was presented in the form that the array offsets can not be equal to a multiple of cache size plus a stencil vector offset.

Following the tool advice we implemented by hand a number of changes in original OpenMP code. Almost all changes were performed in `rhs`, `zsolve`, `buts`, and `blts` subroutines. We classify these changes into 3 categories as shown in Table 2: removing auxiliary arrays and nest fusion in `rhs`, loop interchange in `rhs`, removing auxiliary arrays in solvers: `zsolve` in BT and SP and `buts`, `blts` in LU.

Both `exact_rhs` and `erhs` were outside of the main iteration loop so we did not do any changes in these subroutines. An incremental improvement in performance via data traffic optimization for each benchmark is tabulated in Table 2. The total improvement was about of factor of 3 for `rhs` and 20% for `zsolve` resulting in overall speedup about 27% on 16 processors.

Table 2: *Improving Benchmark Performance via Data Traffic Optimization.* Time (in seconds) was measured on 16 processors of 400MHz Origin 2000 machine.

Application	Original Code	Data reuse and nest fuse	Nest interchange	Removing aux. arrys	Total speedup
BT.A	54.00	51.44	44.22	42.12	22%
SP.A	63.40	55.89	38.70	37.92	40%
LU.A	59.35	52.19	48.95	48.78	18%

Some optimizations, which give performance improvements in simple test codes did not result in expected improvements in the benchmarks.

• Optimizing page size by providing each processor with one page from each array, did not effect performance of the jobs running under PBS. It improved performance of jobs running outside PBS by 10%.

• We could not find a remedy for thread interference warning (high CI number) in `zsolve`. The nest had dependency in z-direction, preventing us from loop interchange. The other option, array transposition, itself involves an interference which has CI number comparable with that of the original code.

5 Conclusions and Related Work

We presented a tool for automatic analysis of data traffic problems in array oriented Fortran codes. The tool advises the user on excessive cache and TLB misses, possible thread interference and suggests the code transformations for improving data traffic. Some of the

transformations are contraintuitive since they reduce data traffic and overall computational time by increasing of the number of floating point instructions.

We showed efficiency of the data traffic improvements on example of three simulated CFD applications BT, SP and LU from NAS Parallel Benchmark suit. For some subroutines working in z-direction the transformations have improved performance by factor of 3 and changed scalability of these subroutines. Overall data traffic optimizations improved the benchmark performance over 27% in average on 16 processors.

Potentially, ADAPT can provide information about filling of the secondary cache after each nest. This information can be used for further improvement of the application performance by a seamless reusing data in the secondary cache between nests and subroutines. We have plans to implement this and some other global data traffic control features in ADAPT and as well as to test it on a wider class of CFD applications.

Research on data traffic control is actively performed in three main directions: reducing communications in MPI programs see [8], optimizing data distributions in HPF programs, see a survey in [2] and improving spatial and temporal data locality for optimizing cache performance [4]. With proliferation of DSM architecture (4 processor DSM may often be spotted at engineers desks) and of OpenMP standard the data traffic control on DSM moves into focus of this research. Some problems of data distributions and page migrations on DSM are subject of recent papers [9, 10]. We expect that with growing depth of computer memory hierarchy and memory-to-processor gap the optimization of the data traffic control will be essential component of program development for DSM architectures.

References

[1] D. Bailey. *NAS Parallel Benchmarks.* http://www.nas.nasa.gov, PBN are available upon request.

[2] M. Frumkin, J. Yan. *Automatic Data Distribution for CFD Applications on Structured Grids.* The 3rd Annual HPF User Group Meeting, Redondo Beach, CA, August 1-2, 1999, 5 pp. Full version: NAS Technical report NAS-99-012, December 99. 27 pp.

[3] M. Frumkin, R.F. Van der Wijngaart. *Efficient Cache Use for Stencil Operations on Structured Discretization Grids.* Submitted to JACM, see also NAS Technical Report NAS-00-15, http://www.nas.nasa.gov/Research/Reports/techreports.html.

[4] S. Gosh, M. Martonosi, S. Malik. *Cache Miss Equations: An Analytical Representation of Cache Misses.* ACM ICS 1997, pp. 317–324.

[5] J.L. Hennessy, D.A. Patterson. *Computer Organization and Design.* Morgan Kaufmann Publishers, San Mateo, CA, 1994.

[6] Innovative Computing Lab., UTK. *PAPI: Hardware Performance Counter Application Programming Interface.* http://icl.cs.utk.edu/papi.

[7] H. Jin, M. Frumkin, J. Yan. *The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance*, NAS Technical Report, NAS-99-011, 1999, 26 pp, http://www.nas.nasa.gov/Research/Reports/Techreports/1999/1999.html.

[8] S.P. Johnson, C.S. Ierotheou, M. Cross. *Automatic Parallel Code Generation on Distributed Memory Systems.* Parallel Computing, V. 22 (1996), pp. 227-258.

[9] D. S. Nikolopoulos, T. S. Papatheodorou, C. D. Polychrnopoulos, J. Labrta, E. Ayguade. *Is Data Distribution Necessary in OpenMP?* Proceedings of Supercomputing 2000. Dallas, TX, Nov. 4-10, 2000. 14 pp.

[10] D. S. Nikolopoulos, T. S. Papatheodorou, C. D. Polychrnopoulos, J. Labrta, E. Ayguade. *Leveraging Transparent Data Distribution in OpenMP via User-Level Dynamic Page Migration.* Springer LNCS, v. 1940, p. 415–427.

[11] *Proceedings of Supercomputing 2000.* Dallas, TX, Nov.4-10, 2000.

[12] SGI Inc. Technical Document. *Origin2000 and Onyx2 Performance Tuning and Optimization Guide.* http://techpubs.sgi.com.

[13] J. Taft. *Performance of the Overflow-MLP CFD Code on the NASA Ames 512 CPU Origin System.* NAS Thechnical Report, NAS-00-05. http://www.nas.nasa.gov/Research/Reports/techreports.html.